Searching an Array:
Linear and Binary Search

Searching

- **Check if a given element (key) occurs in the array.**
- **Two methods to be discussed:**
  - **If the array elements are unsorted.**
    - **Linear search**
  - **If the array elements are sorted.**
    - **Binary search**

Linear Search

- **Basic idea:**
  - **Start at the beginning of the array.**
  - **Inspect every element to see if it matches the key.**
- **Time complexity:**
  - **A measure of how long an algorithm takes to run.**
  - **If there are n elements in the array:**
    - **Best case:**

      match found in first element (1 search operation)
    - **Worst case:**

      no match found, or match found in the last element (n search operations)
    - **Average:**

      $(n + 1) / 2$ search operations

Contd.

```
/* If key appears in a[0..size-1], return its location, pos, s.t.
   a[pos] == key. If key is not found, return -1 */
int  linear_search (int a[], int size, int key)
{
        int pos = 0;
        while ((pos < size) && (a[pos] != key))
                pos++;
        if (pos<n)
                return pos;     /* Return the position of match */
        return -1;              /* No match found */
}
```

Contd.

int x[ ]= {12, -3, 78, 67, 6, 50, 19, 10} ;

- **Trace the following calls :**

  search (x, 8, 6) ;

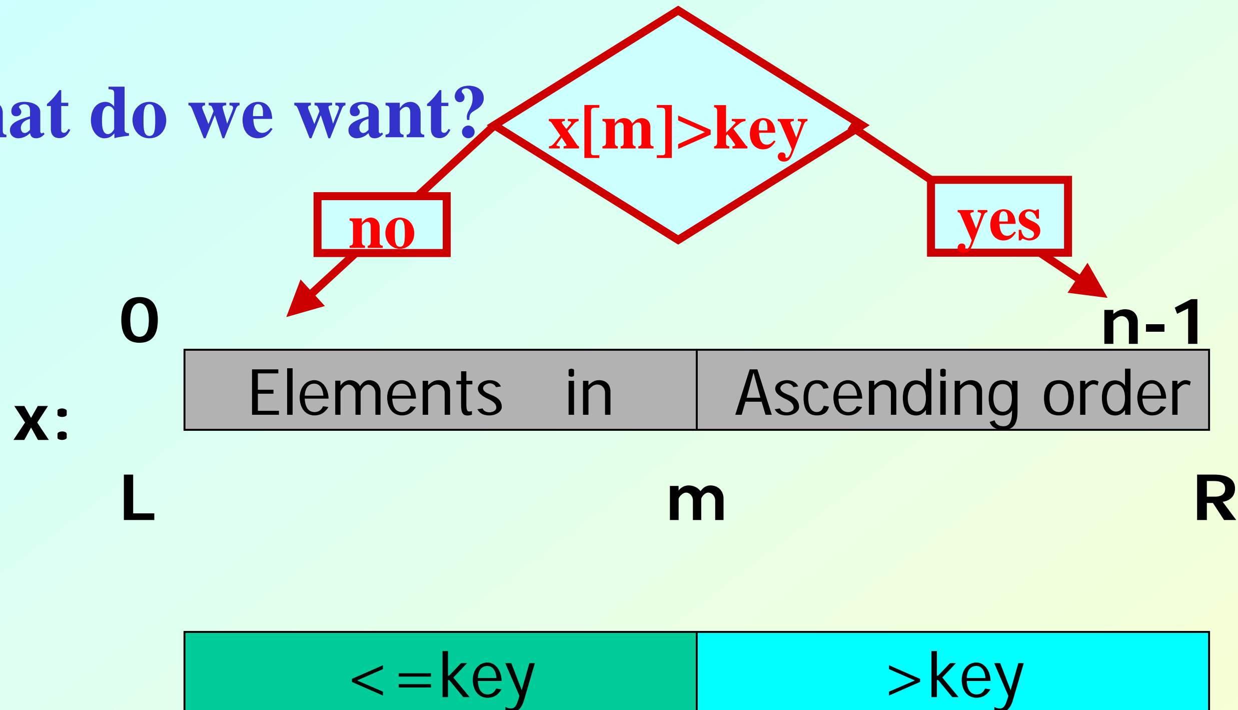
  search (x, 8, 5) ;

Binary Search

- Binary search works if the array is **sorted**.
  - Look for the target in the middle.
  - If you don't find it, you can ignore half of the array, and repeat the process with the other half.
- In every step, we reduce the number of elements to search in by half.

The Basic Strategy

- What do we want?



  - Look at [(L+R)/2]. Move L or R to the middle depending on test.
  - Repeat search operation in the reduced interval.

Contd.

```
/* If key appears in x[0..size-1], return its location, pos s.t.
    x[pos]==key. If not found, return -1 */

int  bin_search (int x[], int size, int key)
{
    int L, R, mid;
    _________________;
    while ( ____________ )
    {
        __________________;
    }
    _________________ ;
}
```

The basic search iteration

```
/* If key appears in x[0..size-1], return its location, pos s.t. x[pos]==key. If
     not found, return -1 */

int  bin_search (int x[], int size, int key)
{
     int L, R, mid;
     __________________;
     while ( _____________ )
     {
          mid = (L + R) / 2;
          if  (x[mid] > key)
               R = mid;
          else L = mid;
     }
     __________________ ;
}
```

Loop termination

```
/* If key appears in x[0..size-1], return its location, pos s.t. x[pos]==key. If
     not found, return -1 */

int  bin_search (int x[], int size, int key)
{
     int L, R, mid;
     _________________;
     while ( L+1 != R )
     {
          mid = (L + R) / 2;
          if  (x[mid] <= key)
               L = mid;
          else R = mid;
     }
     _________________ ;
}
```

Return result

```
/* If key appears in x[0..size-1], return its location, pos s.t. x[pos]==key. If
     not found, return -1 */

int  bin_search (int x[], int size, int key)
{
     int L, R, mid;
     _________________;
     while ( L+1 != R )
     {
               mid = (L + R) / 2;
               if  (x[mid] <= key)
                    L = mid;
               else R = mid;
     }
     if   (L >= 0  &&  x[L] = = key)   return L;
     else  return  -1;
}
```

Initialization

```
/* If key appears in x[0..size-1], return its location, pos s.t. x[pos]==key. If
    not found, return -1 */

int  bin_search (int x[], int size, int key)
{
    int L, R, mid;
    L = -1;   R = size;
    while ( L+1 != R )
    {
        mid = (L + R) / 2;
        if  (x[mid] <= key)
            L = mid;
        else R = mid;
    }
    if   (L >= 0  &&  x[L] = = key)   return L;
    else  return  -1;
}
```

Binary Search Examples

Sorted array

-17	-5	3	6	12	21	45	63	50

Trace :

binsearch (x, 9, 3); →

L= -1; R= 9; x[4]=12;
L= -1; R=4; x[1]= -5;
L= 1; R=4; x[2]=3;
L=2; R=4; x[3]=6;
L=2; R=3; return L;

binsearch (x, 9, 145);

binsearch (x, 9, 45);

We may modify the algorithm by checking equality with x[mid].

Is it worth the trouble ?

- Suppose there are 1000 elements.
- Ordinary search
- If key is a member of x, it would require 500 comparisons on the average.
- Binary search
  - after 1st compare, left with 500 elements.
  - after 2nd compare, left with 250 elements.
  - After at most 10 steps, you are done.

Time Complexity

- If there are n elements in the array.
  - Number of searches required:

    $\log_2 n$

- For n = 64 (say).
  - Initially, list size = 64.
  - After first compare, list size = 32.
  - After second compare, list size = 16.
  - After third compare, list size = 8.
  - ....
  - After sixth compare, list size = 1.

$2^k = n$,

Where k is the number of steps.

$\log_2 64 = 6$

Sorting: the basic problem

- Given an array

  x[0], x[1], ... , x[size-1]

  reorder entries so that

  x[0] <= x[1] <= . . . <= x[size-1]

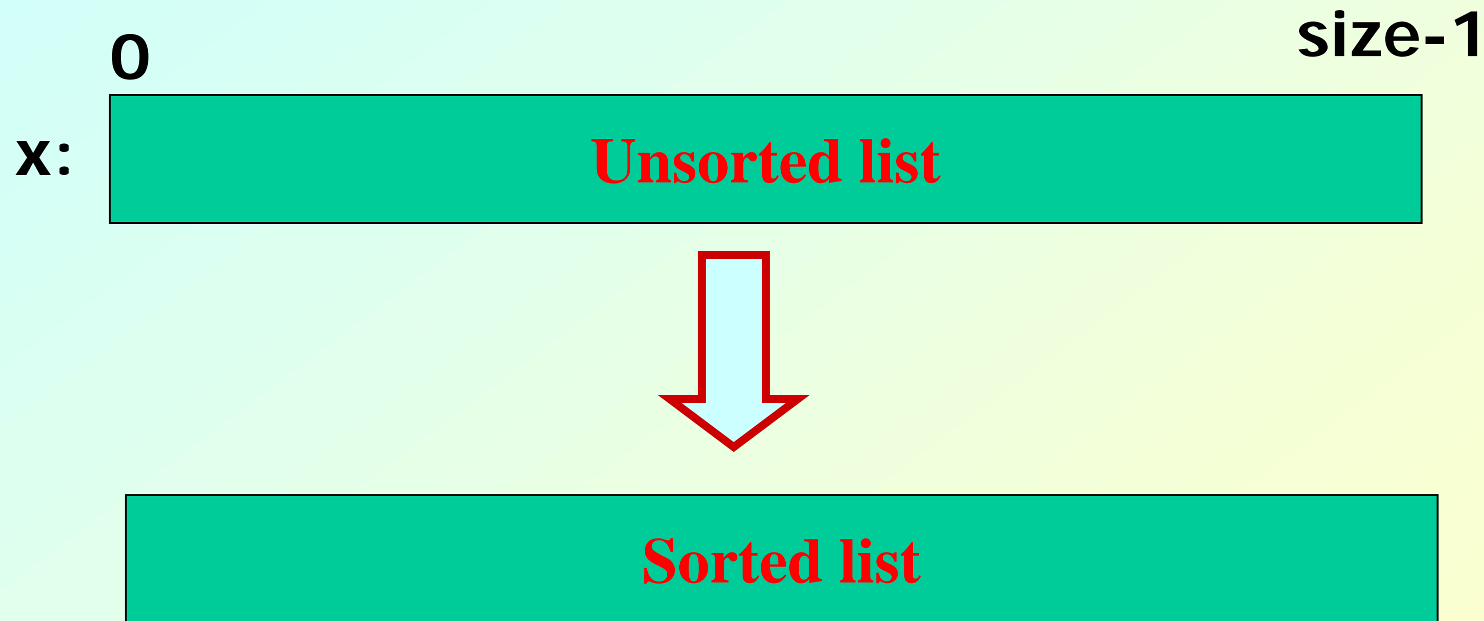
  - List is in non-decreasing order.
- We can also sort a list of elements in non-increasing order.

Example

- **Original list:**
  - **10, 30, 20, 80, 70, 10, 60, 40, 70**
- **Sorted in non-decreasing order:**
  - **10, 10, 20, 30, 40, 60, 70, 70, 80**
- **Sorted in non-increasing order:**
  - **80, 70, 70, 60, 40, 30, 20, 10, 10**

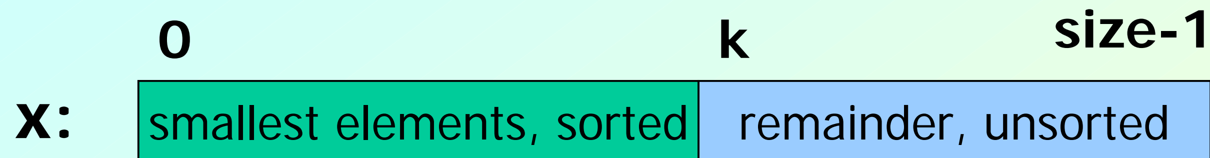
Sorting Problem

- **What we want : Data sorted in order**

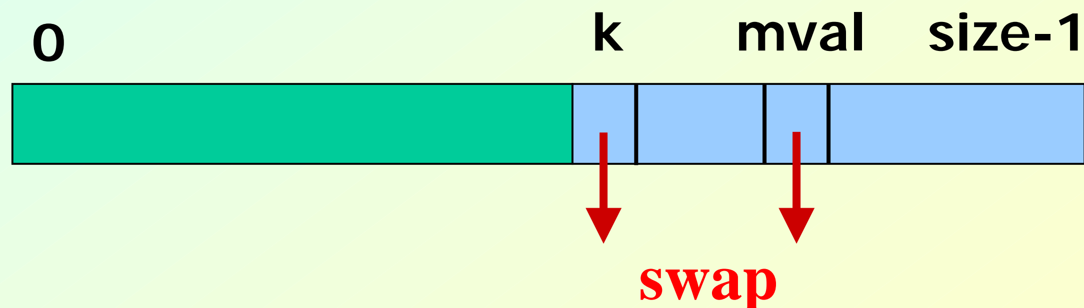


Selection Sort

- **General situation :**



- **Step :**
  - **Find smallest element, mval, in x[k..size-1]**
  - **Swap smallest element with x[k], then increase k.**



Subproblem

```
/* Yield location of smallest element in x[k .. size-1];*/

int min_loc (int x[ ], int k, int size)
{
    int j, pos;      /* x[pos] is the smallest element found so far */
    pos = k;
    for (j=k+1; j<size; j++)
        if (x[j] < x[pos])
            pos = j;
    return pos;
}
```

The main sorting function

```
/* Sort x[0..size-1] in non-decreasing order */

int selsort (int x[], int size)
{  int k, m;
    for (k=0; k<size-1; k++)
    {
            m = min_loc(x, k, size);
            temp = a[k];
            a[k] = a[m];
            a[m] = temp;
    }
}
```

Example

x:	3	12	-5	6	142	21	-17	45

x:	-17	12	-5	6	142	21	3	45

x:	-17	-5	12	6	142	21	3	45

x:	-17	-5	3	6	142	21	12	45

x:	-17	-5	3	6	142	21	12	45

x:	-17	-5	3	6	12	21	142	45

x:	-17	-5	3	6	12	21	142	45

x:	-17	-5	3	6	12	21	45	142

Analysis

- **How many steps are needed to sort n things ?**
  - **Total number of steps proportional to n^2**
  - **No. of comparisons?**

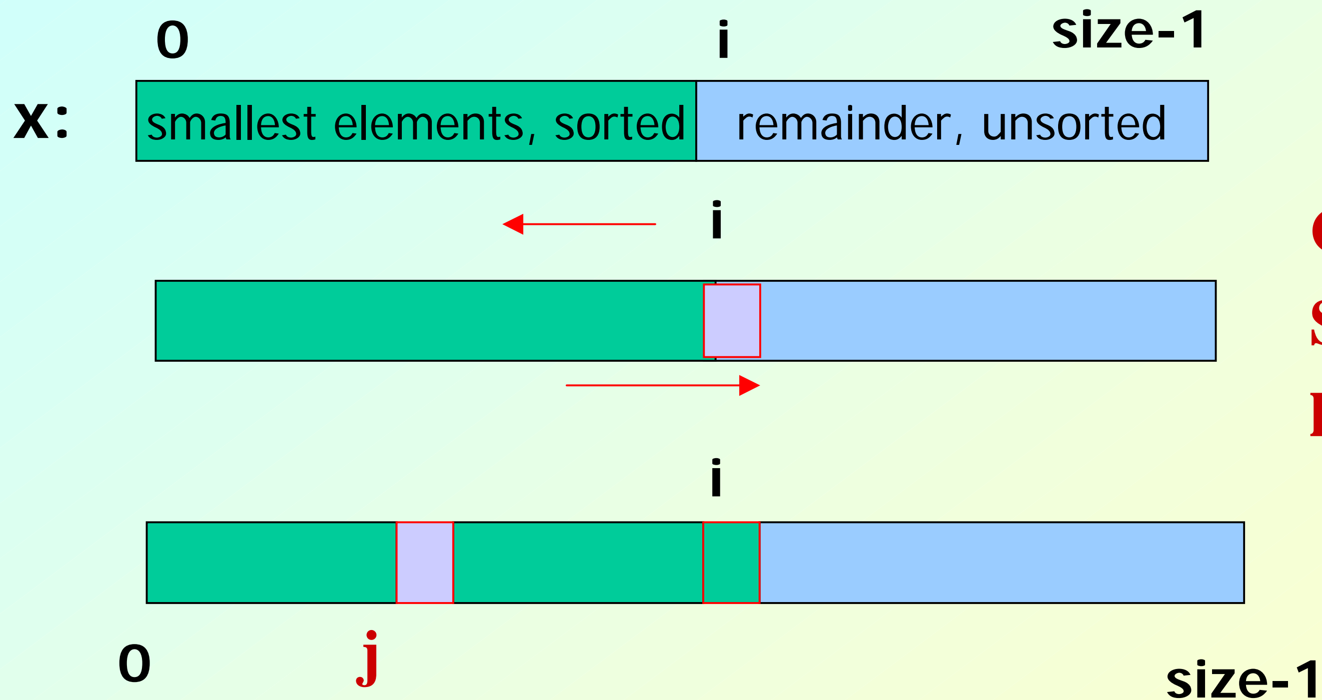
$$(n-1)+(n-2)+......+1= n(n-1)/2$$

Of the order of n^2

  - **Worst Case? Best Case? Average Case?**

Insertion Sort

- **General situation :**



Compare and Shift till x[i] is larger.

Insertion Sorting

```
void InsertSort (int list[], int size)
{
    for (i=1; i<size; i++)
    {
        item = list[i] ;
        for (j=i-1; (j>=0)&& (list[j] > i); j--)
            list[j+1] = list[j];
        list[j+1] = item ;
    }
}
```

Insertion Sort

```
#define MAXN 100
void InsertSort (int list[MAXN], int size) ;
main ()  {
    int index, size;
    int numbers[MAXN];
    /* Get Input */
    size = readarray (numbers) ;
    printarray (numbers, size) ;
    InsertSort (numbers, size) ;
    printarray (numbers, size) ;
}
```

Time Complexity

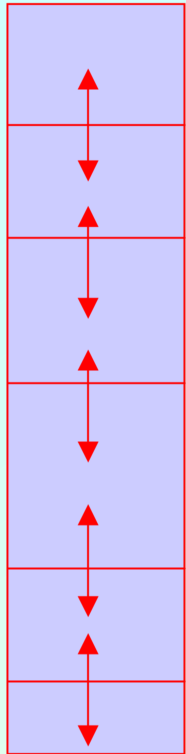
- **Number of comparisons and shifting:**
  - **Worst Case?**

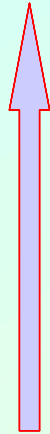
$1+2+3+ \ldots\ldots +(n-1) = n(n-1)/2$

  - **Best Case?**

$1+1+\ldots\ldots (n-1)$ times $= (n-1)$

Bubble Sort





In every iteration heaviest element drops at the bottom.

The bottom moves upward.

Bubble Sort

```
#include <stdio.h>

void swap(int *x,int *y)
{
  int tmp=*x;
  *x=*y;
  *y=tmp;
}
```

```
void bubble_sort(int x[],int n)
{
  int i,j;

 for(i=n-1;i>0;i--)
 for(j=0;j<i;j++)
  if(x[j]>x[j+1]) swap(&x[j],&x[j+1]);

}
```

Contd.

```
int main(int argc, char *argv[])
{
  int x[ ]={-45,89,-65,87,0,3,-23,19,56,21,76,-50};
  int i;
  for(i=0;i<12;i++) printf("%d ",x[i]);
   printf("\n");
   bubble_sort(x,12);
  for(i=0;i<12;i++)   printf("%d ",x[i]);
   printf("\n");
}
```

-45 89 -65 87 0 3 -23 19 56 21 76 -50

-65 -50 -45 -23 0 3 19 21 56 76 87 89

Time Complexity

- **Number of comparisons :**
  - **Worst Case?**

    $1+2+3+ \ldots\ldots +(n-1) = n(n-1)/2$

  - **Best Case?**

    same.

How do you make best case with (n-1) comparisons only?

Some Efficient Sorting Algorithms

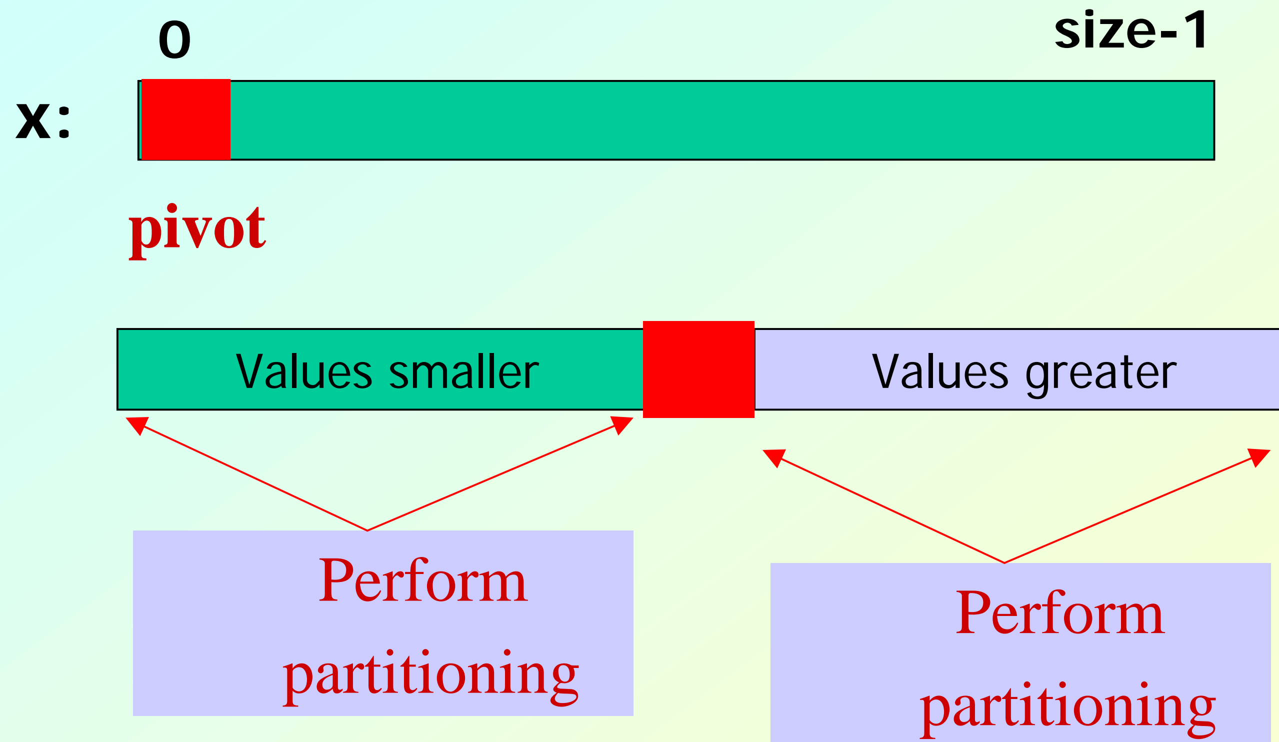
- **Two of the most popular sorting algorithms are based on divide-and-conquer approach.**
  - **Quick sort**
  - **Merge sort**
- **Basic concept:**

```
sort (list)
{
    if the list has length greater than 1
    {
        Partition the list into lowlist and highlist;
        sort (lowlist);
        sort (highlist);
        combine (lowlist, highlist);
    }
}
```

Quicksort

- **At every step, we select a pivot element in the list (usually the first element).**
  - **We put the pivot element in the final position of the sorted list.**
  - **All the elements less than or equal to the pivot element are to the left.**
  - **All the elements greater than the pivot element are to the right.**

Partitioning



Quick Sort: Example

```
#include <stdio.h>
#include <stdlib.h>

void print(int x[],int low,int high)
{
  int i;

   for(i=low;i<=high;i++)
   printf(" %d", x[i]);
   printf("\n");
}
```

```
void swap(int *a,int *b)
{
  int tmp=*a;
  *a=*b; *b=tmp;
}
```

Contd.

```
void partition(int x[],int low,int high)
{
 int i=low+1,j=high;
 int pivot=x[low];
 if(low>=high) return;
 while(i<j)
 {
  while ((x[i]<pivot) && (i<high)) i++;
  while ((x[j]>=pivot) && (j>low)) j--;
  if(i<j) swap(&x[i],&x[j]);
 }
if (j==high) {
swap(&x[j],&x[low]);
partition(x,low,high-1);
  }
else
if (i==low+1)
partition(x,low+1,high);
 else {
   swap(&x[j],&x[low]);
   partition(x,low,j-1);
  partition(x,j+1,high);
 }
}
```

Contd:

```
int main(int argc,char *argv[])
{
  int *x;
  int i=0;
  int num;

   num=argc-1;
   x=(int *) malloc(num * sizeof(int));

   for(i=0;i<num; i++)
    x[i]=atoi(argv[i+1]);

   printf("Input: ");
  for(i=0; i<num; i++)
    printf(" %d", x[i]);
    printf("\n");
   partition(x,0,num-1);
   printf("Output: ");
   for(i=0; i<num; i++)
    printf(" %d", x[i]);
    printf("\n");
}
```

Trace of Partitioning

./a.out 45 -56 78 90 -3 -6 123 0 -3 45 69 68

Input: 45 -56 78 90 -3 -6 123 0 -3 45 69 68

45 -56 78 90 -3 -6 123 0 -3 45 69 68

-6 -56 -3 0 -3 45 123 90 78 45 69 68

-56 -6 -3 0 -3 68 90 78 45 69 123

-3 0 -3 45 68 78 90 69

-3 0 69 78 90

Output: -56 -6 -3 -3 0 45 45 68 69 78 90 123

Time Complexity

- **Partitioning with n elements.**

  - No. of comparisons:

  n-1

Choice of pivot element affects the time complexity.

- **Worst Case Performance:**

  (n-1)+(n-2)+(n-3)+………. +1= n(n-1)/2

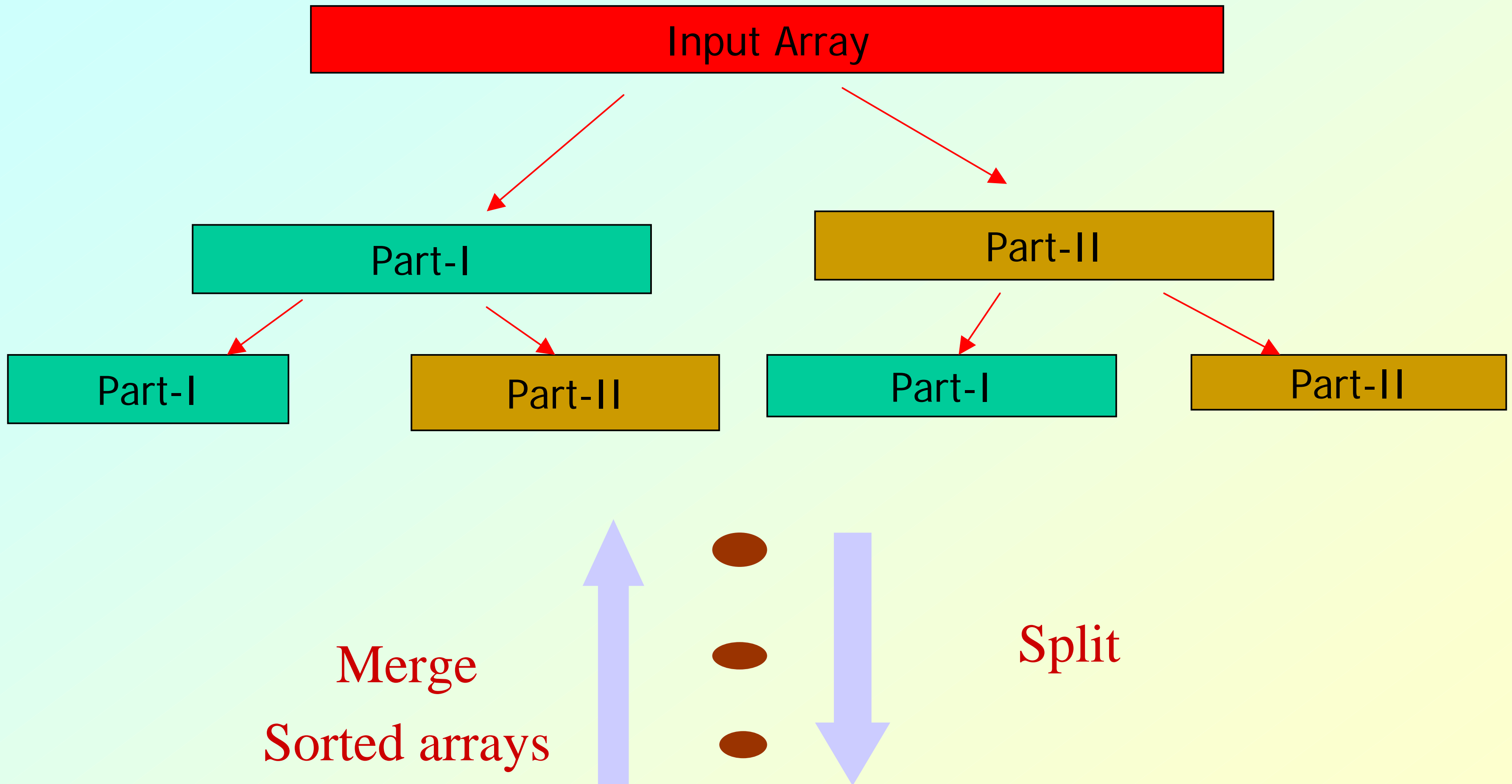
- **Best Case performance:**

  (n-1)+2((n-1)/2-1)+4(((n-1)/2-1)-1)/2-1) .. k steps

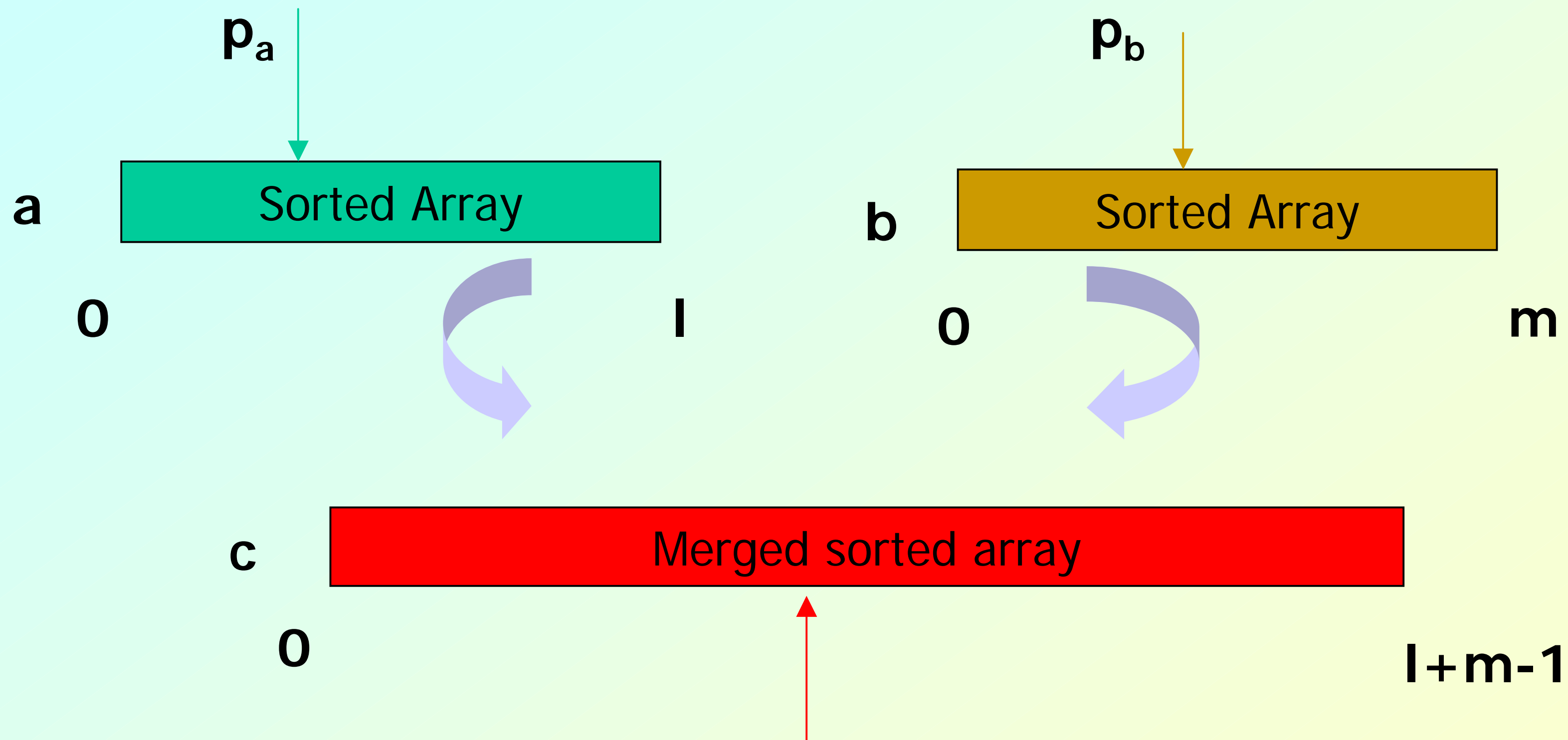
  = O(n. log(n))

$2^k=n$

Merge Sort



Merging two sorted arrays



Move and copy elements pointed by p_a if its value is smaller than the element pointed by p_b in (l+m-1) operations and otherwise.

Merge Sort

```
#include <stdio.h>
#include <stdlib.h>
 main()
{
    int i, num;
    int a[ ]={-56,23,43,-5,-3,0,123,-35,87,56,75,80};
    for(i=0;i<12;i++) printf("%d ",a[i]); printf("\n");
    merge_sort(a, 12);
    printf(“\n The sorted sequence follows \n");
    for(i=0;i<12;i++)  printf("%d ",a[i]); printf("\n");
}
```

/* Recursive function for dividing an array a[0..n-1] into two halves and sort them and merge them subsequently. */

```
void merge_sort(int *a,int n)
{
int i,j,l,m;
int *b, *c;

if(n>1){
l=n/2; m=n-l;
b=(int *) calloc(l,sizeof(int));
c=(int *) calloc(m,sizeof(int));
for(i=0;i<l;i++) b[i]=a[i];
for(j=l;j<n;j++) c[j-l]=a[j];

merge_sort(b,l);
merge_sort(c,m);
merge(b,c,a,l,m);
free(b); free(c);
}
}
```

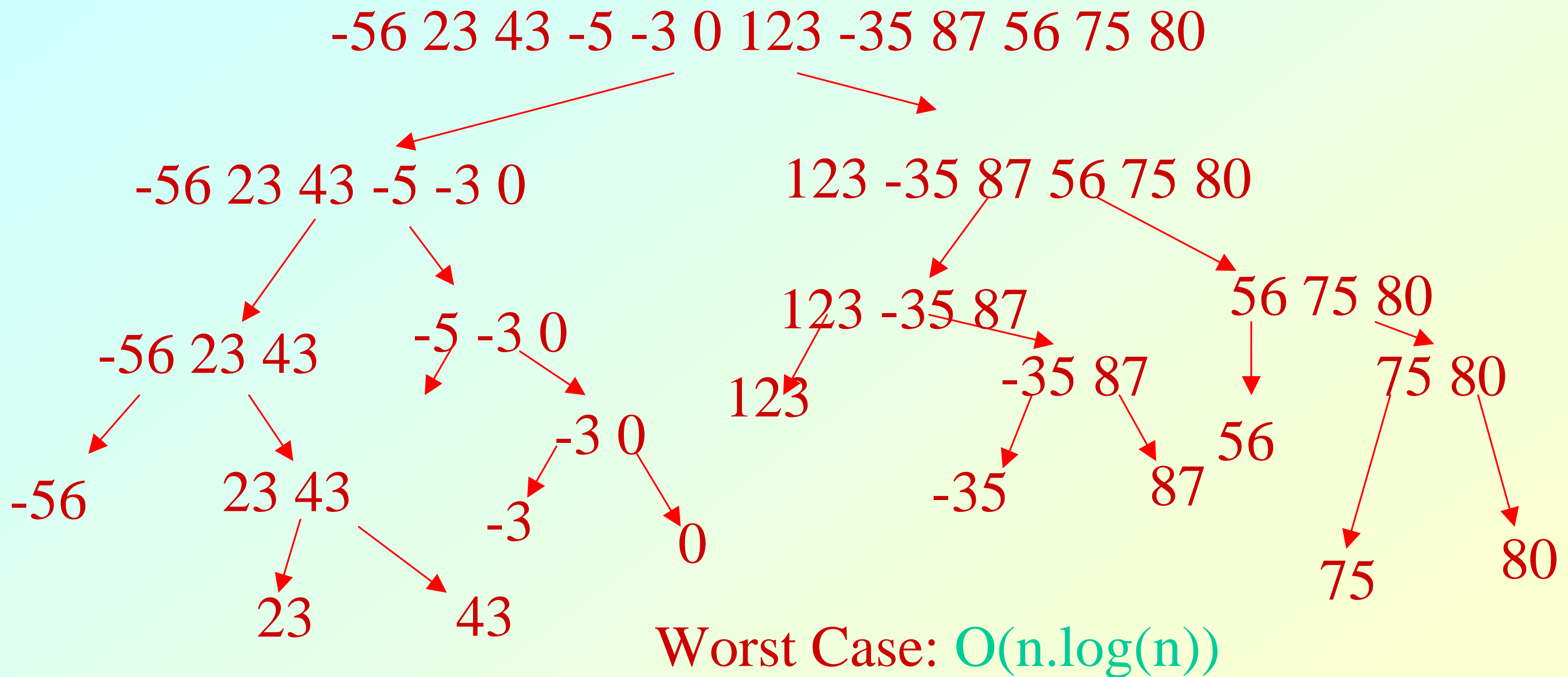
/* Merging of two sorted arrays a[0..m-1] and b[0..n-1] into a single array c[0..m+n-1] */

```
void merge(int *a,int *b,int *c,
                    int m,int n)
{int i,j,k,l;

i=j=k=0;

do{
if(a[i]<b[j]){ c[k]=a[i]; i=i+1; }
else{ c[k]=b[j]; j=j+1;}
k++;
} while((i<m)&&(j<n));
```

```
if(i==m){
for(l=j;l<n;l++){ c[k]=b[l]; k++;}
}
else{
for(l=i;l<m;l++){c[k]=a[l]; k++;}
}
}
```

Pointer movement and copy operations.

Splitting Trace



Worst Case: O(n.log(n))

-56 -35 -5 -3 0 23 43 56 75 80 87 123